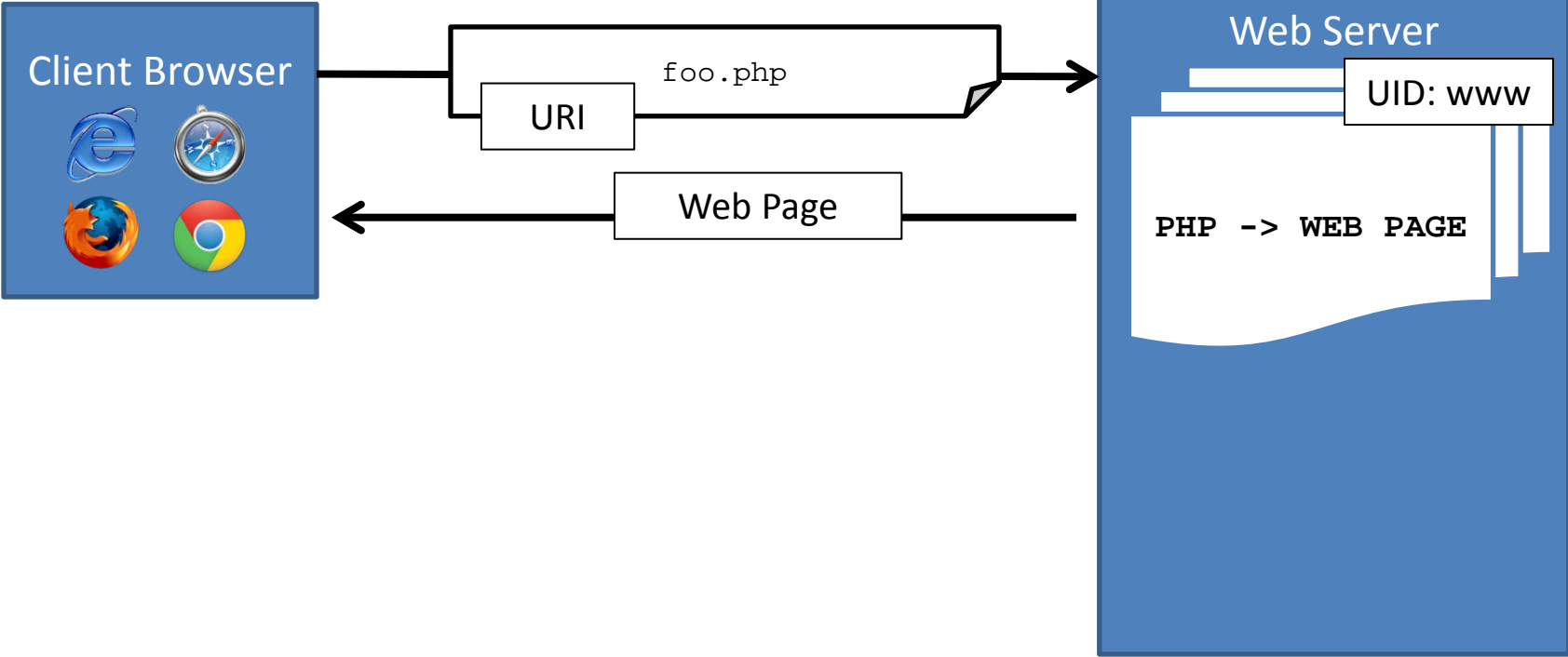


Web Security: Vulnerabilities & Attacks

Command Injection

Background



Quick Background on PHP

```
display.php: <? echo system("cat ".$_GET['file']); ?>
```

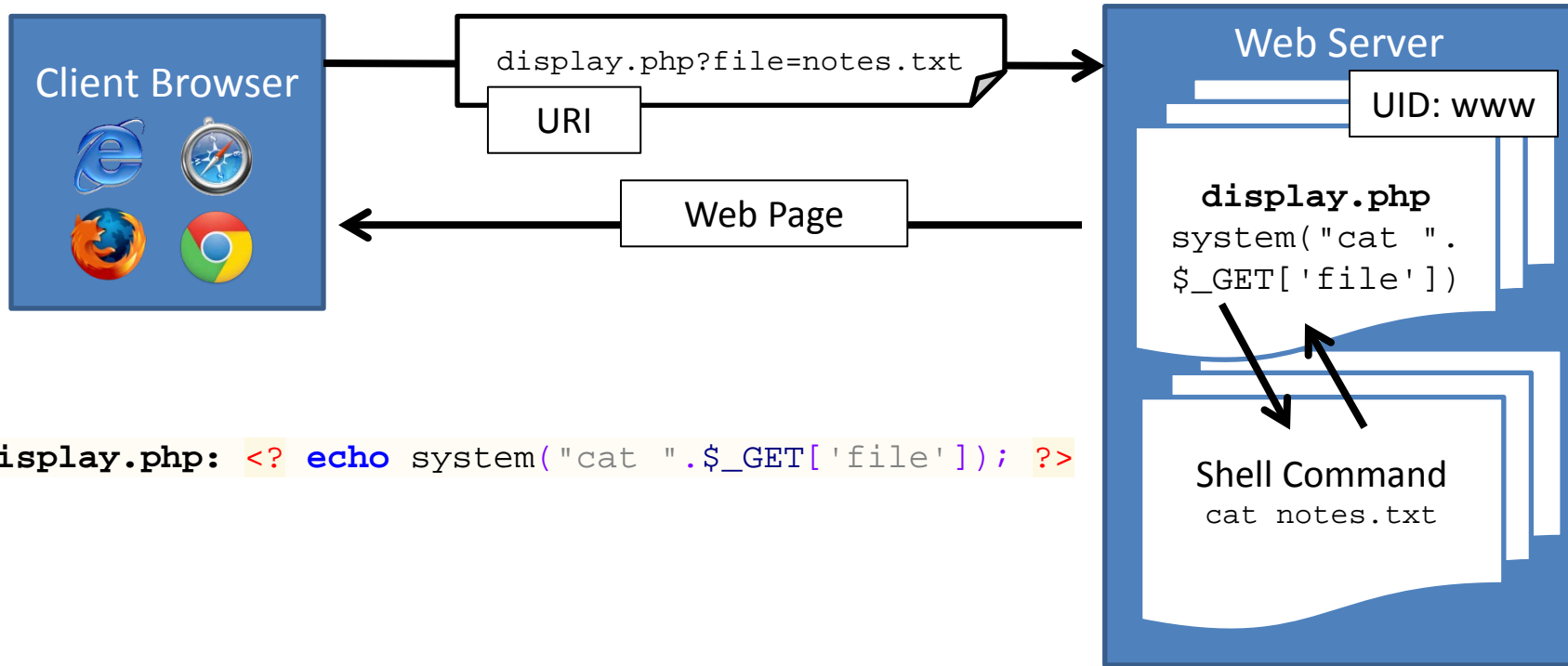
IN THIS EXAMPLE

| | |
|--|--|
| <code><? <i>php-code</i> ?></code> | executes php-code at this point in the document |
| <code>echo expr:</code> | evaluates expr and embeds in doc |
| <code>system(call, args)</code> | performs a system call in the working directory |
| <code>"", '</code> | String literal. Double-quotes has more possible escaped characters. |
| <code>.</code> | (dot). Concatenates strings. |
| <code>\$_GET['key']</code> | returns <i>value</i> corresponding to the <i>key/value</i> pair sent as extra data in the HTTP GET request |

LATER IN THIS LECTURE

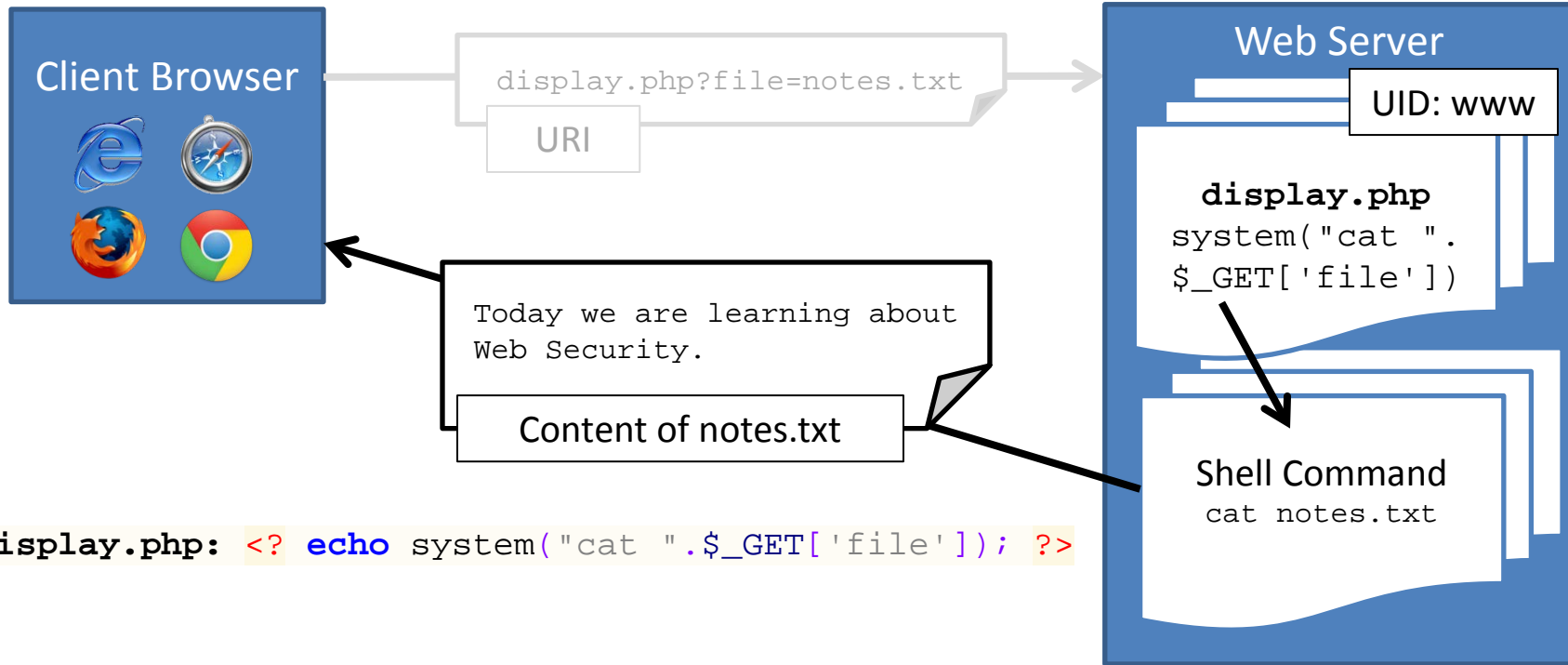
| | |
|---|---|
| <code>preg_match(Regex, Stiring)</code> | Performs a regular expression match. |
| <code>proc_open</code> | Executes a command and opens file pointers for input/output. |
| <code>escapeshellarg()</code> | Adds single quotes around a string and quotes/escapes any existing single quotes. |
| <code>file_get_contents(file)</code> | Retrieves the contents of file. |

Background



```
display.php: <? echo system("cat " . $_GET['file']); ?>
```

Background



Command Injection

```
display.php: <? echo system("cat ".$_GET['file']); ?>
```

Q: Assuming the script we've been dealing with (reproduced above) for `http://www.example.net/display.php`. Which one of the following URIs is an attack URI?

Hint: Search for a URI Decoder to figure out values seen by the PHP code.

- `http://www.example.net/display.php?get=rm`
- `http://www.example.net/display.php?file=rm%20-rf%20%2F%3B`
- `http://www.example.net/display.php?file=notes.txt%3B%20rm%20-rf%20%2F%3B%0A%0A`
- `http://www.example.net/display.php?file=%20%20%20%20%20`



Command Injection

```
display.php: <? echo system("cat ".$_GET['file']); ?>
```

Q: Assuming the script we've been dealing with (reproduced above) for `http://www.example.net/display.php`. Which one of the following URIs is an attack URI?

Hint: Search for a URI Decoder to figure out values seen by the PHP code.

(URIs decoded)

- a. `http://www.example.net/display.php?get=rm`
- b. `http://www.example.net/display.php?file=rm -rf /;`
- c. `http://www.example.net/display.php?file=notes.txt; rm -rf /;`
- d. `http://www.example.net/display.php?file=`



Command Injection

```
display.php: <? echo system("cat ".$_GET['file']); ?>
```

Q: Assuming the script we've been dealing with (reproduced above) for `http://www.example.net/display.php`. Which one of the following URIs is an attack URI?

Hint: Search for a URI Decoder to figure out values seen by the PHP code.

(Resulting php)

- a. `<? echo system("cat rm"); ?>`
- b. `<? echo system("cat rm -rf /;"); ?>`
- c. `<? echo system("cat notes.txt; rm -rf /;"); ?>`
- d. `<? echo system("cat "); ?>`



Injection

- Injection is a general problem:
 - Typically, caused when data and code share the same *channel*.
 - For example, the code is “*cat*” and the filename the data.
 - But ‘*;*’ allows attacker to start a new command.

Input Validation

- Two forms:
 - Blacklisting: Block known attack values
 - Whitelisting: Only allow known-good values
- Blacklists are easily bypassed
 - Set of ‘attack’ inputs is potentially infinite
 - The set can change after you deploy your code
 - Only rely on blacklists as a part of a defense in depth strategy

Blacklist Bypass

| Blacklist | Bypass |
|--|---|
| Disallow semi-colons | Use a pipe |
| Disallow pipes and semi colons | Use the backtick operator to call commands in the arguments |
| Disallow pipes, semi-colons, and backticks | Use the \$ operator which works similar to backtick |
| Disallow rm | Use unlink |
| Disallow rm, unlink | Use cat to overwrite existing files |

- *Ad infinitum*
- Tomorrow, newer tricks might be discovered

Input Validation: Whitelisting

display.php:

```
<?
if(!preg_match("/^[a-z0-9A-Z.]*$/", $_GET['file'])) {
    echo "The file should be alphanumeric.";
    return;
}
echo system("cat " . $_GET['file']);
?>
```

| GET INPUT | PASSES? |
|----------------------|---------|
| notes.txt | Yes |
| notes.txt; rm -rf /; | No |
| security notes.txt | No |

Input Escaping

display.php:

```
<?
#http://www.php.net/manual/en/function.escapeshellarg.php
echo system("cat ".escapeshellarg($_GET['file']));
?>
```

escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument

```
-- http://www.php.net/manual/en/function.escapeshellarg.php
```

| GET INPUT | Command Executed |
|----------------------|---------------------------|
| notes.txt | cat 'notes.txt' |
| notes.txt; rm -rf /; | cat 'notes.txt rm -rf /;' |
| mary o'donnel | cat 'mary o'\''donnel' |

Use less powerful API

- The system command is too powerful
 - Executes the string argument in a new shell
 - If only need to read a file and output it, use simpler API
- ```
display.php: <? echo file_get_contents($_GET['file']); ?>
```
- Similarly, the *proc\_open* (executes commands and opens files for I/O) API
    - Can only execute one command at a time.

# Recap

- Command Injection: a case of *injection*, a general vulnerability
- Defenses against injection include input validation, input escaping and use of a less powerful API
- Next, we will discuss other examples of injection and apply similar defenses



# SQL Injection

# Background

- SQL: A query language for database
  - E.g., `SELECT` statement, `WHERE` clauses
- More info
  - E.g., <http://en.wikipedia.org/wiki/SQL>

# Running Example

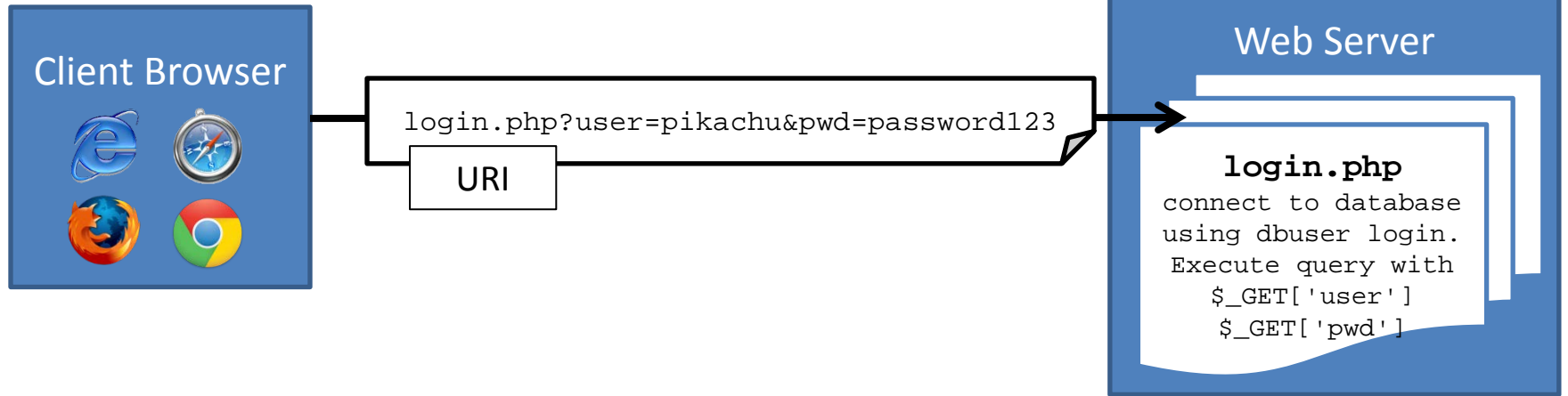
Consider a web page that logs in a user by seeing if a user exists with the given username and password.

`login.php:`

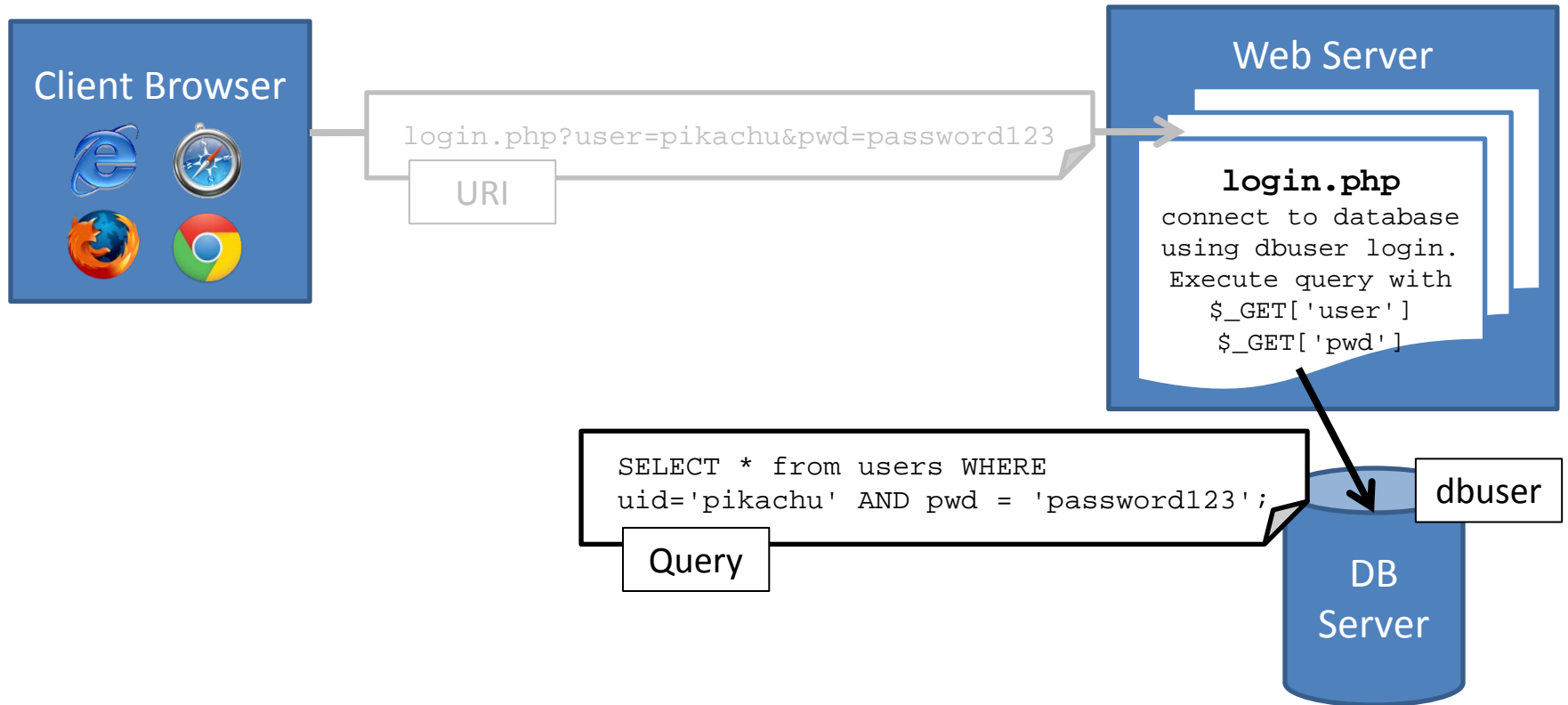
```
$result = pg_query("SELECT * from users WHERE
 uid = '$_GET['user'].'" AND
 pwd = '$_GET['pwd'].'"");
if (pg_query_num($result) > 0) {
 echo "Success";
 user_control_panel_redirect();
}
```

It sees if results exist and if so logs the user in and redirects them to their user control panel.

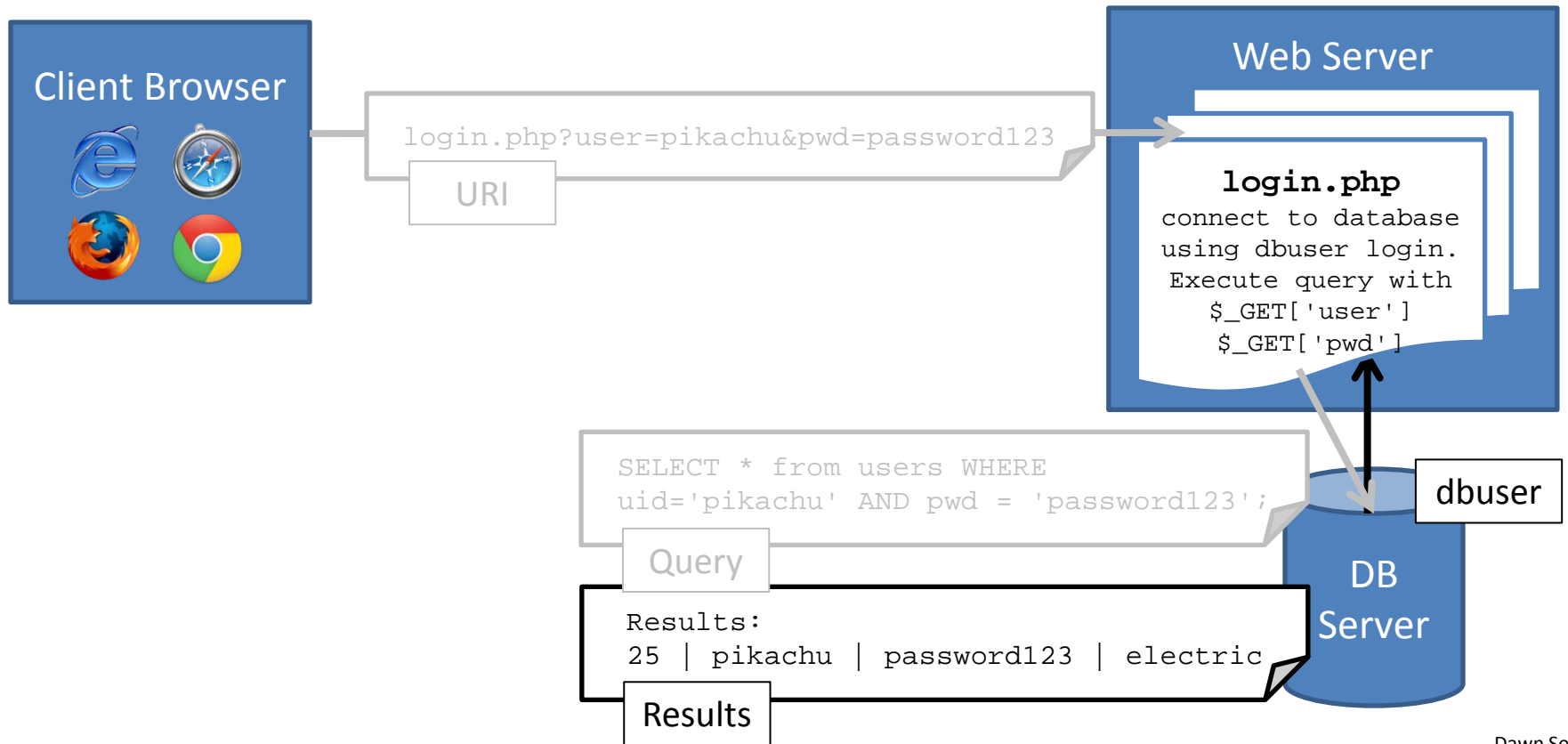
# Background



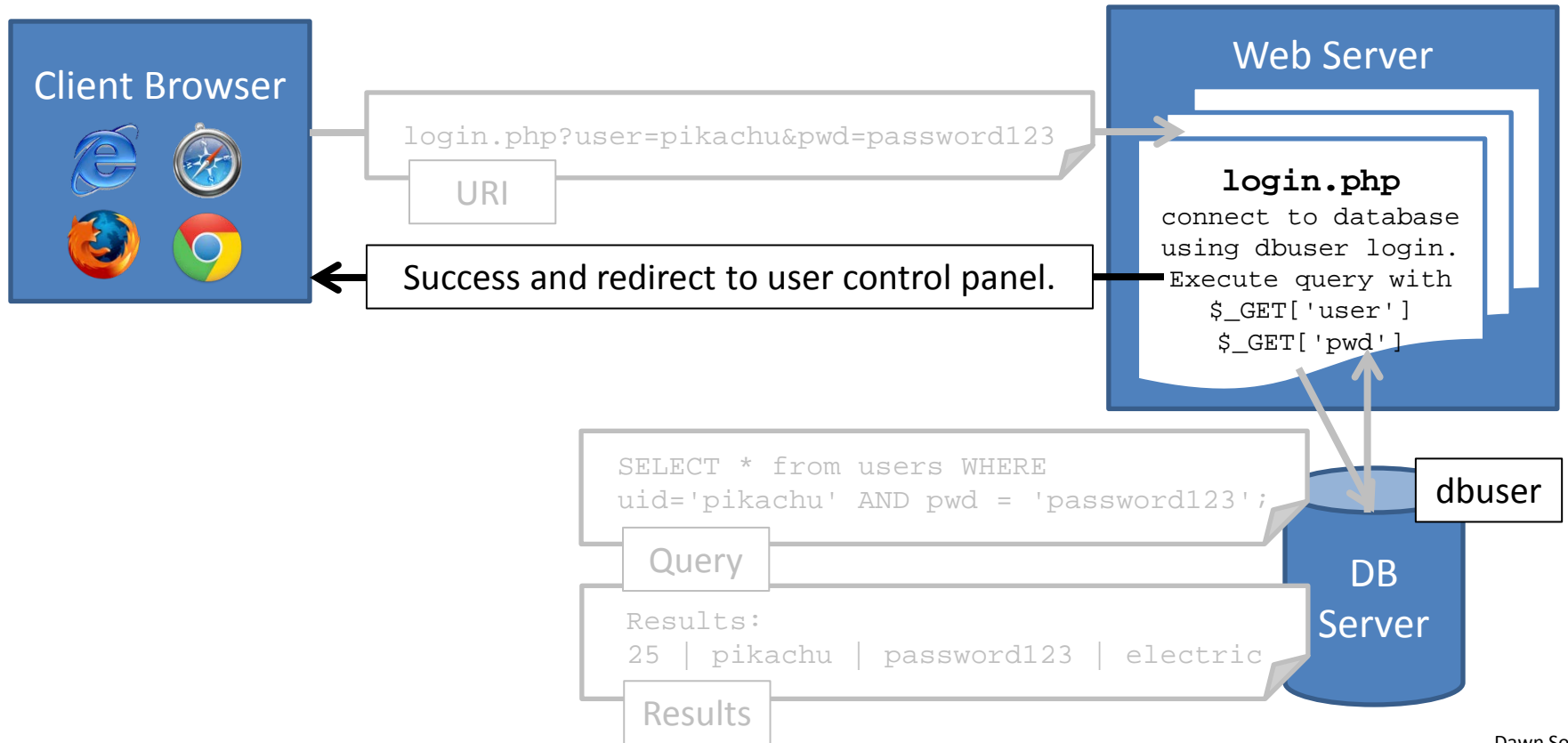
# Background



# Background



# Background



# SQL Injection

login.php:

```
$result = pg_query("SELECT * from users WHERE
 uid = '". $_GET['user'] ."' AND
 pwd = '". $_GET['pwd'] ."'");
if (pg_query_num($result) > 0) {
 echo "Success";
 user_control_panel_redirect();
}
```

Q: Which one of the following queries will log you in as admin?

Hints: The SQL language supports comments via '--' characters.

- a. `http://www.example.net/login.php?user=admin&pwd='`
- b. `http://www.example.net/login.php?user=admin--&pwd=foo`
- c. `http://www.example.net/login.php?user=admin'--&pwd=f`



# SQL Injection

login.php:

```
$result = pg_query("SELECT * from users WHERE
 uid = '". $_GET['user'] ."' AND
 pwd = '". $_GET['pwd'] ."'");
if (pg_query_num($result) > 0) {
 echo "Success";
 user_control_panel_redirect();
}
```

Q: Which one of the following queries will log you in as admin?

Hints: The SQL language supports comments via '--' characters.

- a. `http://www.example.net/login.php?user=admin&pwd='`
- b. `http://www.example.net/login.php?user=admin--&pwd=foo`
- c. `http://www.example.net/login.php?user=admin'--&pwd=f`



# SQL Injection

login.php:

```
$result = pg_query("SELECT * from users WHERE
 uid = '". $_GET['user'] ."' AND
 pwd = '". $_GET['pwd'] ."'");
if (pg_query_num($result) > 0) {
 echo "Success";
 user_control_panel_redirect();
}
```

URI: <http://www.example.net/login.php?user=admin'--&pwd=f>

```
pg_query("SELECT * from users WHERE
 uid = 'admin'--' AND pwd = 'f'");
```

```
pg_query("SELECT * from users WHERE
 uid = 'admin'");
```

# SQL Injection

Q: Under the same premise as before, which URI can delete the users table in the database?

- a. `www.example.net/login.php?user=;DROP TABLE users;--`
- b. `www.example.net/login.php?user=admin%27%3B%20DROP%20TABLE%20users--%3B&pwd=f`
- c. `www.example.net/login.php?user=admin;%20DROP%20TABLE%20users;%20--&pwd=f`
- d. It is not possible. (None of the above)

# SQL Injection

Q: Under the same premise as before, which URI can delete the users table in the database?

- a. `www.example.net/login.php?user=;DROP TABLE users;--`
- b. `www.example.net/login.php?user=admin'; DROP TABLE users;--&pwd=f` (Decoded)
- c. `www.example.net/login.php?user=admin; DROP TABLE users; --&pwd=f`
- d. It is not possible. (None of the above)

```
pg_query("SELECT * from users WHERE
 uid = 'admin'; DROP TABLE users;--' AND
 pwd = 'f';");
```

```
pg_query("SELECT * from users WHERE uid = 'admin';
 DROP TABLE users;");
```

# SQL Injection

- One of the most exploited vulnerabilities on the web
- Cause of massive data theft
  - 24% of all data stolen in 2010
  - 89% of all data stolen in 2009
- Like command injection, caused when attacker controlled data interpreted as a (SQL) command.



# Injection Defenses

- Defenses:
  - Input validation
    - Whitelists untrusted inputs to a safe list.
  - Input escaping
    - Escape untrusted input so it will not be treated as a command.
  - Use less powerful API
    - Use an API that only does what you want
    - Prefer this over all other options.



# Input Validation for SQL

login.php:

```
<?
if(!preg_match("/^[a-z0-9A-Z.]*$/", $_GET['user'])) {
 echo "Username should be alphanumeric.";
 return;
}
// Continue to do login query
?>
```

| GET INPUT                    | PASSES? |
|------------------------------|---------|
| Pikachu                      | Yes     |
| Pikachu'; DROP TABLE users-- | No      |
| O'Donnell                    | No      |

# Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URLs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
 uid = '" . $_GET['user'] . "' AND
 pwd = '" . $_GET['pwd'] . "' ;");
```

- a. `http://www.example.net/login.php?user=admin&pwd=admin`
- b. `http://www.example.net/login.php?user=admin&pwd='%20OR%201%3D1;--`
- c. `http://www.example.net/login.php?user=admin'--&pwd=f`
- d. `http://www.example.net/login.php?user=admin&pwd='--`



# Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URLs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
 uid = '" . $_GET['user'] . "' AND
 pwd = '" . $_GET['pwd'] . "'");
```

- a. <http://www.example.net/login.php?user=admin&pwd=admin>
- b. <http://www.example.net/login.php?user=admin&pwd='%20OR%201%3D1;-->
- c. <http://www.example.net/login.php?user=admin'--&pwd=f>
- d. <http://www.example.net/login.php?user=admin&pwd='-->



# Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URIs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
 uid = '$_GET['user']' AND
 pwd = '$_GET['pwd']';");
```

b. <http://www.example.net/login.php?user=admin&pwd=' OR 1=1;-->

```
pg_query("SELECT * from users WHERE
 uid = 'admin' AND
 pwd = ' OR 1 = 1;--';");
```



# Input Validation for SQL

Given that our web application employs the input validation mechanism for usernames, which of the following URLs would still allow you to login as admin?

```
pg_query("SELECT * from users WHERE
 uid = '" . $_GET['user'] . "' AND
 pwd = '" . $_GET['pwd'] . "';");
```

```
pg_query("SELECT * from users WHERE
 (uid = 'admin' AND pwd = '') OR
 1 = 1;--'");
```

1=1 is true everywhere. This returns all the rows in the table, and thus number of results is greater than zero.

# Input Escaping

```
$_GET['user'] = pg_escape_string($_GET['user']);
$_GET['pwd'] = pg_escape_string($_GET['pwd']);
```

***pg\_escape\_string()*** escapes a string for querying the PostgreSQL database. It returns an escaped literal in the PostgreSQL format.

| GET INPUT                  | Escaped Output              |
|----------------------------|-----------------------------|
| Bob                        | Bob                         |
| Bob'; DROP TABLE users; -- | Bob''; DROP TABLE users; -- |
| Bob' OR '1'='1             | Bob'' OR ''1''=''1          |

# Use less powerful API :

## Prepared Statements

- Create a template for SQL Query, in which data values are substituted.
- The *database* ensures untrusted value isn't interpreted as command.
- Always prefer over all other techniques.
- Less powerful:
  - Only allows queries set in templates.



# Use less powerful API : Prepared Statements

```
<?
The $1 and $2 are a 'hole' or place holder for what will be filled by the data
$result = pg_query_params('SELECT * FROM users WHERE
 uid = $1 AND
 pwd = $2', array($_GET['user'], $_GET['pwd']));

Compare to
$result = pg_query("SELECT * FROM users WHERE
 uid = '$_GET['user']' AND
 pwd = '$_GET['pwd']'");
?>
```

# Recap

- SQL Injection: a case of *injection*, in database queries.
- Extremely common, and pervasively exploited.
- Use prepared statements to prevent SQL injection
  - **DO NOT** use escaping, despite what xkcd says.
- Next, injection in the browser.

# Cross-site Scripting



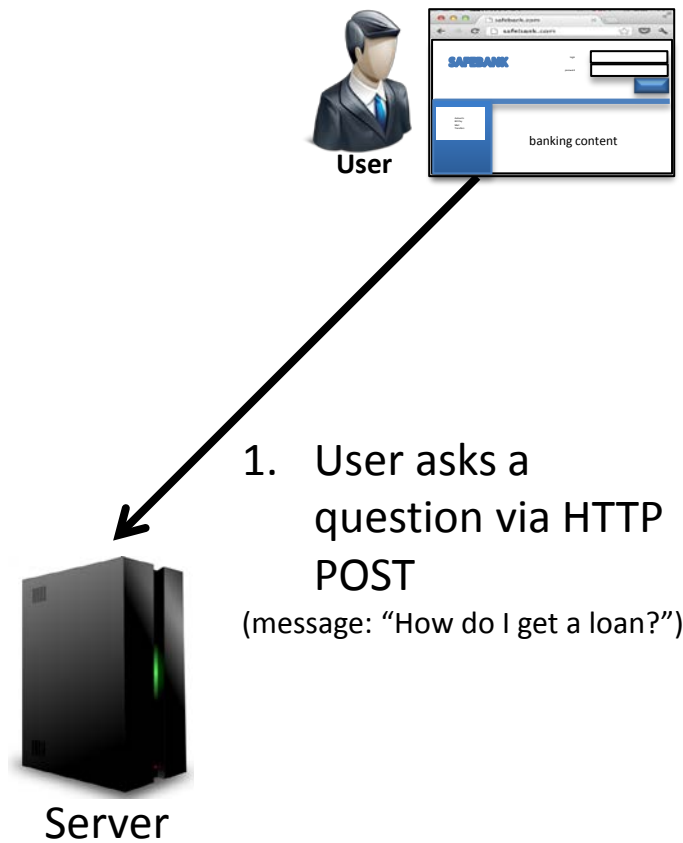
# What is Cross-site Scripting (XSS)?

- Vulnerability in web application that enables attackers to inject client-side scripts into web pages viewed by other users.

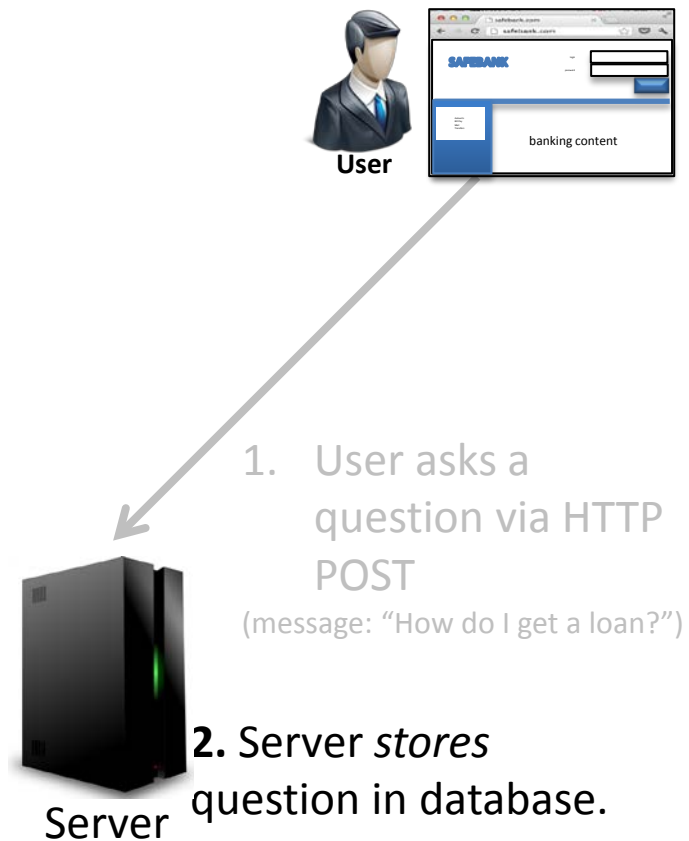
# Three Types of XSS

- **Type 2: Persistent or Stored**
  - **The attack vector is stored at the server**
- Type 1: Reflected
  - The attack value is 'reflected' back by the server
- Type 0: DOM Based
  - The vulnerability is in the client side code

Consider a form on [safebank.com](http://safebank.com) that allows a user to chat with a customer service associate.



Consider a form on [safebank.com](http://safebank.com) that allows a user to chat with a customer service associate.



Consider a form on [safebank.com](http://safebank.com) that allows a user to chat with a customer service associate.



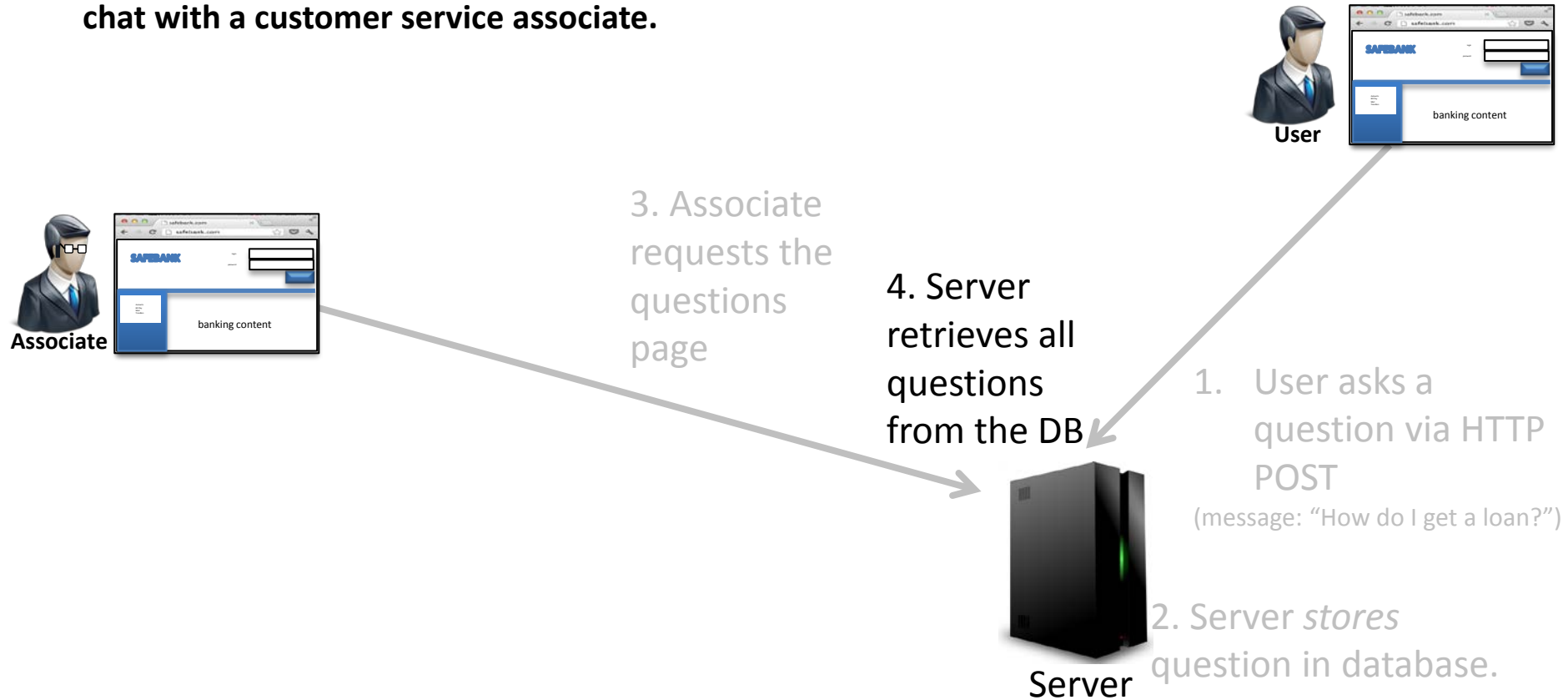
3. Associate requests the questions page



1. User asks a question via HTTP POST  
(message: "How do I get a loan?")

2. Server stores question in database.

Consider a form on [safebank.com](http://safebank.com) that allows a user to chat with a customer service associate.



```

PHP CODE: <? echo "<div class='question'>$question</div>" ;?>
HTML Code: <div class='question'>"How do I get a loan?"</div>

```



User



Associate

3. Associate requests the questions page

4. Server retrieves all questions from the DB

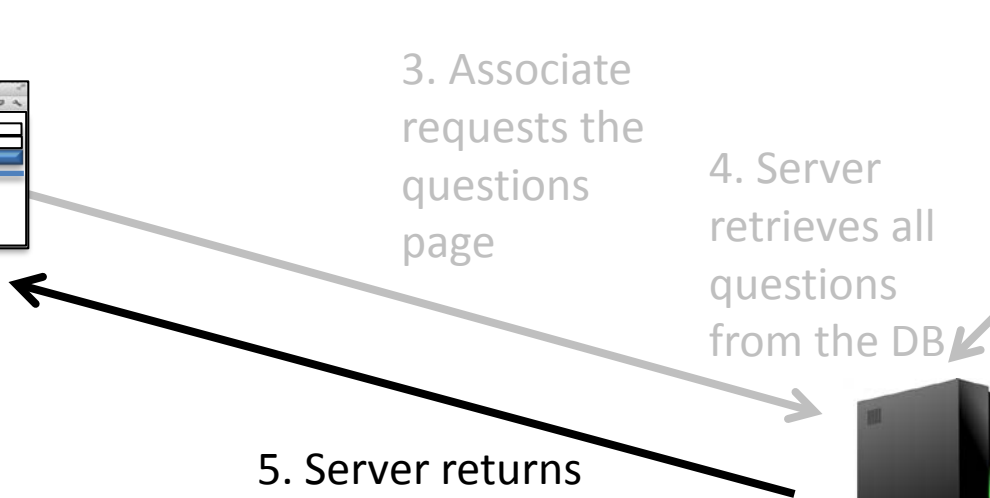
1. User asks a question via HTTP POST  
(message: "How do I get a loan?")

2. Server stores question in database.

5. Server returns HTML embedded with the question



Server



```

PHP CODE: <? echo "<div class='question'>$question</div>";?>
HTML Code: <div class='question'>"How do I get a loan?"</div>

```



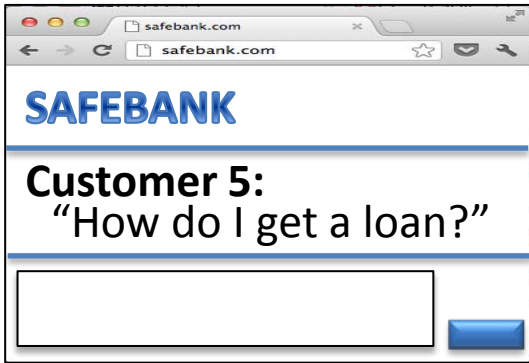
1. User asks a question via HTTP POST (message: "How do I get a loan?")

2. Server stores question in database.

4. Server retrieves all questions from the DB

3. Associate requests the questions page

5. Server returns HTML embedded with the question





# Type 2 XSS Injection

Look at the following code fragments. Which one of these could possibly be a comment that could be used to perform a XSS injection?

- a. `' ; system('rm -rf /');`
- b. `rm -rf /`
- c. `DROP TABLE QUESTIONS;`
- d. `<script>doEvil()</script>`

# Script Injection

Which one of these could possibly be a comment that could be used to perform a XSS injection?

- a. `' ; system('rm -rf /');`
- b. `rm -rf /`
- c. `DROP TABLE QUESTIONS;`
- d. `<script>doEvil()</script>`

```
<html><body>
 ...
 <div class='question'>
 <script>doEvil()</script>
 </div>
 ...
</body></html>
```

# Stored XSS



# Stored XSS



1. Attacker asks malicious question via HTTP POST  
( message: "`<script>doEvil()</script>`" )



Server

2. Server stores question in database.

# Stored XSS



3. Victim requests the questions page



Server

1. Attacker asks malicious question via HTTP POST ( message: "<script>doEvil(</script>") )

2. Server stores question in database.

# Stored XSS



3. Victim requests the questions page

4. Server retrieves malicious question from the DB

1. Attacker asks malicious question via HTTP POST (message: "<script>doEvil(</script>")

2. Server stores question in database.



# Stored XSS

```
PHP CODE: <? echo "<div class='question'>$question</div>" ;?>
```

```
HTML Code: <div class='question'><script>doEvil()</script></div>
```



3. Victim requests the questions page

4. Server retrieves malicious question from the DB

1. Attacker asks malicious question via HTTP POST (message: "<script>doEvil()</script>")

2. Server stores question in database.

5. Server returns HTML embedded with malicious question



Server

# Stored XSS

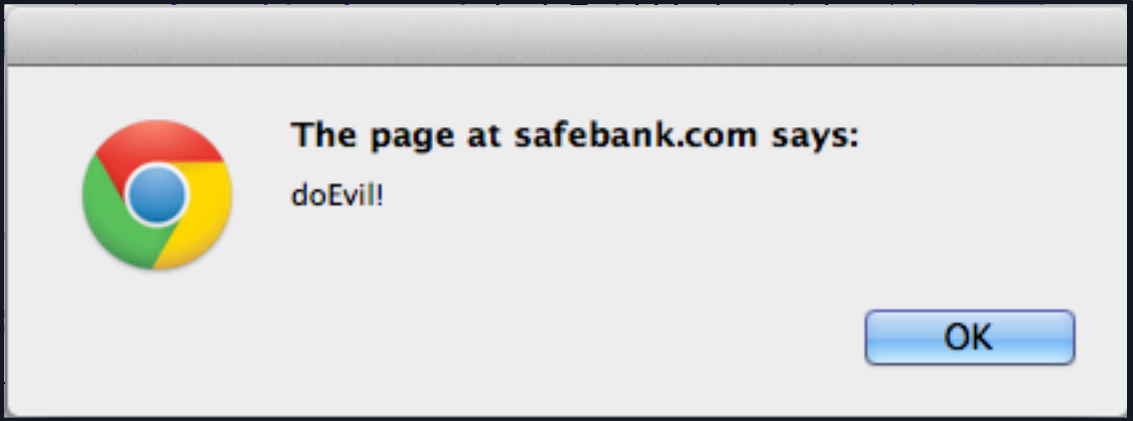
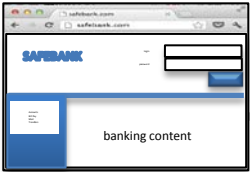


```
PHP CODE: <? echo "<div class='question'>$question</div>" ;?>
```

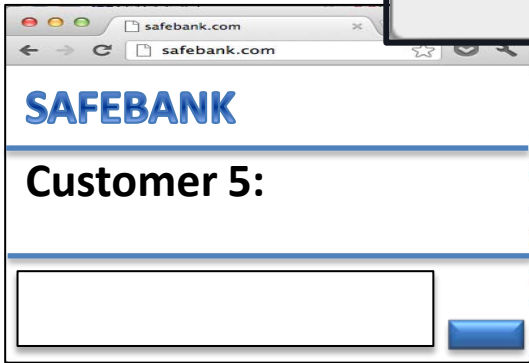
```
HTML Code: <div class='question'>doEvil!</div>
```



Associate



hacker asks malicious question via HTTP POST (message: "<script>doEvil()</script>")



5. Server returns HTML embedded with malicious question



Server

2. Server stores question in database.



# Three Types of XSS

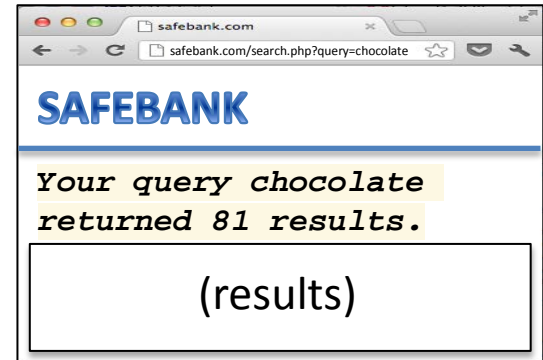
- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- **Type 1: Reflected**
  - **The attack value is 'reflected' back by the server**
- Type 0: DOM Based
  - The vulnerability is in the client side code

# Example Continued: Blog

- safebank.com also has a transaction search interface at search.php
- search.php accepts a query and shows the results, with a helpful message at the top.

```
<? echo "Your query $_GET['query'] returned $num results." ;?>
```

**Example: *Your query chocolate returned 81 results.***



- What is a possible malicious URI an attacker could use to exploit this?

# Type 1: Reflected XSS

A request to “search.php?query=<script>doEvil()</script>” causes script injection. Note that the query is never stored on the server, hence the term 'reflected'

PHP Code: `<? echo "Your query $_GET['query'] returned $num results." ;?>`

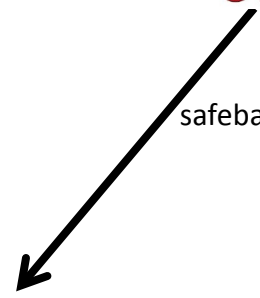
HTML Code: `Your query <script>doEvil()</script> returned 0 results`

But this only injects code in the attacker's page. The attacker needs to make the user click on this link, for the attack to be effective.

# Reflected XSS



1. Send Email with malicious link  
`safebank.com/search.php?query=<script>doEvil()</script>`



Vulnerable Server

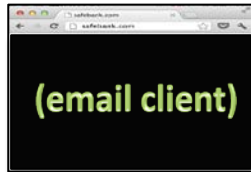
# Reflected XSS



1. Send Email  
with malicious link  
`safebank.com/search.php?query=<script>doEvil()</script>`



User



2. Click on Link with malicious params



Vulnerable Server

# Reflected XSS



1. Send Email with malicious link  
 safebank.com/search.php?query=<script>doEvil()</script>



2. Click on Link with malicious params

```
Your query
<script>doEvil()</script>
returned 0 results
```

3. Server inserts malicious params into HTML



Vulnerable Server

# Reflected XSS



1. Send Email with malicious link  
`safebank.com/search.php?query=<script>doEvil()</script>`

```
Your query
<script>doEvil()</script>
returned 0 results
```

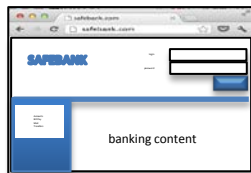
2. Click on Link with malicious params

3. Server inserts malicious params into HTML

4. HTML with injected attack code

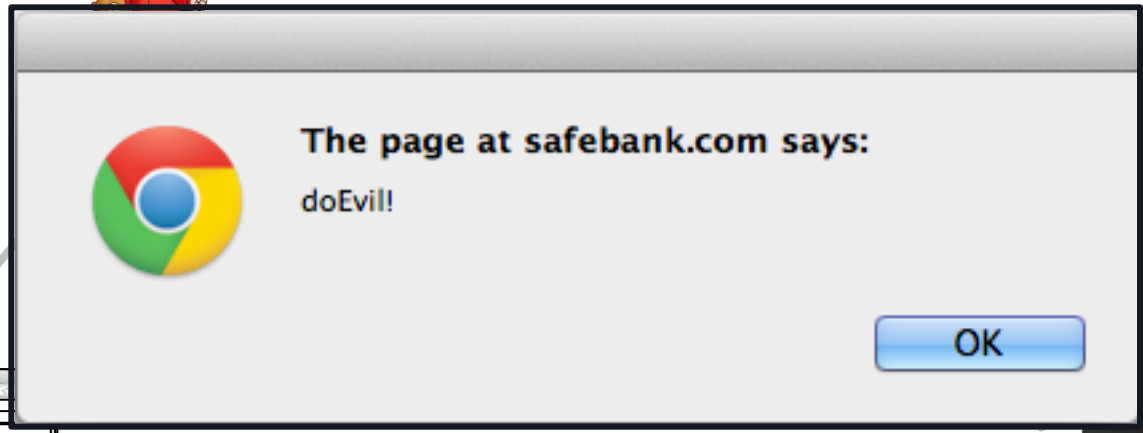


User



Vulnerable Server

# Reflected XSS



```
query
<script>doEvil()</script>
0 results
```

inserts malicious  
into HTML



User



4. HTML with injected attack code



Vulnerable Server

5. Execute embedded  
malicious script.



# Three Types of XSS

- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- Type 1: Reflected
  - The attack value is 'reflected' back by the server
- **Type 0: DOM Based**
  - **The vulnerability is in the client side code**

# Type 0: Dom Based XSS

- Traditional XSS vulnerabilities occur in the *server side code*, and the fix involves improving sanitization at the server side.
- Web 2.0 applications include significant processing logic, at the client side, written in JavaScript.
- Similar to the server, this code can also be vulnerable.
- When the XSS vulnerability occurs in the client side code, it is termed as a DOM Based XSS vulnerability

# Type 0: Dom Based XSS

Suppose safebank.com uses client side code to display a friendly welcome to the user. For example, the following code shows “Hello Joe” if the URL is

`http://safebank.com/welcome.php?name=Joe`

**Hello**

```
<script>
```

```
var pos=document.URL.indexOf("name=")+5;
```

```
document.write(document.URL.substring(pos,document.URL.length));
```

```
</script>
```

# Type 0: Dom Based XSS

For the same example, which one of the following URIs will cause untrusted script execution?

Hello

```
<script>
```

```
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

- a. `http://attacker.com`
- b. `http://safebank.com/welcome.php?name=doEvil()`
- c. `http://safebank.com/welcome.php?name=<script>doEvil()</script>`

# Type 0: Dom Based XSS

For the same example, which one of the following URIs will cause untrusted script execution?

Hello

```
<script>
```

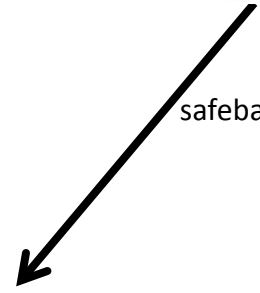
```
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

- a. `http://attacker.com`
- b. `http://safebank.com/welcome.php?name=doEvil()`
- c. `http://safebank.com/welcome.php?name=<script>doEvil()</script>`

# DOM-Based XSS



1. Send Email with malicious link  
`safebank.com/welcome.php?query=<script>doEvil()</script>`



Vulnerable Server

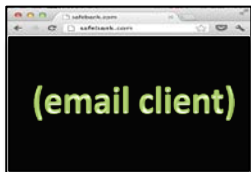
# DOM-Based XSS



1. Send Email  
with malicious link  
`safebank.com/welcome.php?query=<script>doEvil()</script>`



User

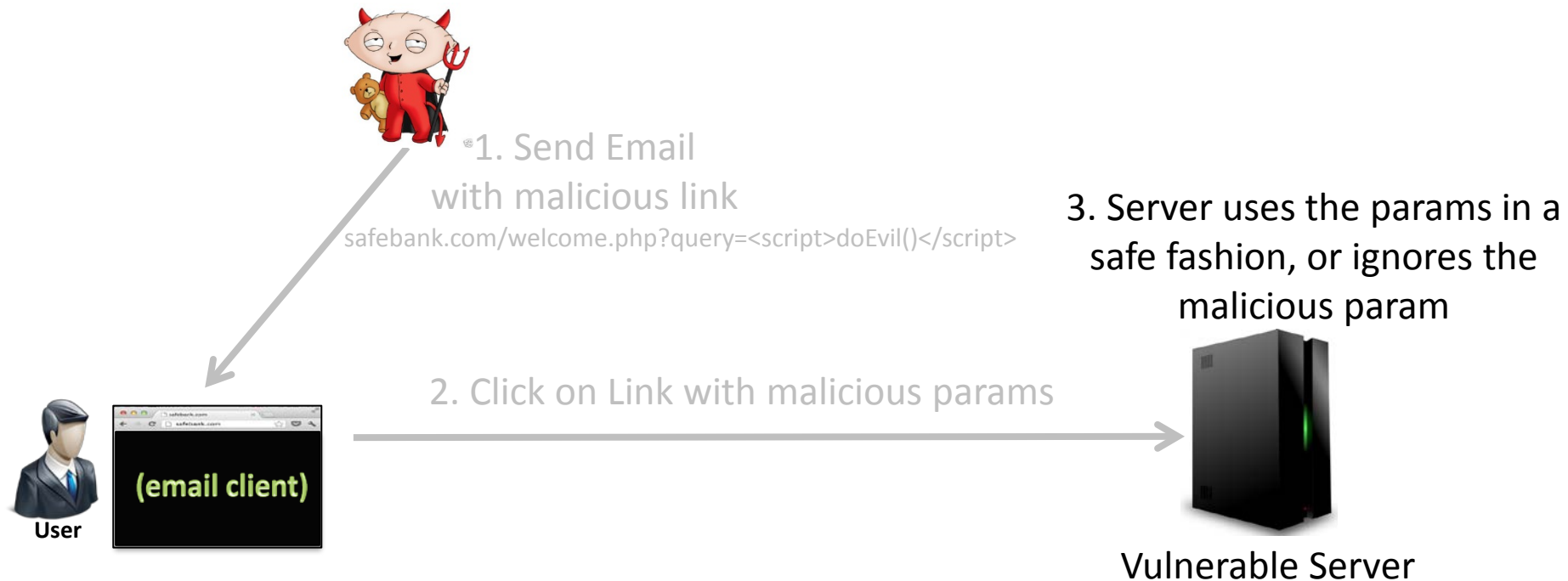


2. Click on Link with malicious params



Vulnerable Server

# DOM-Based XSS





# DOM-Based XSS



1. Send Email with malicious link  
 safebank.com/welcome.php?query=<script>doEvil()</script>



2. Click on Link with malicious params

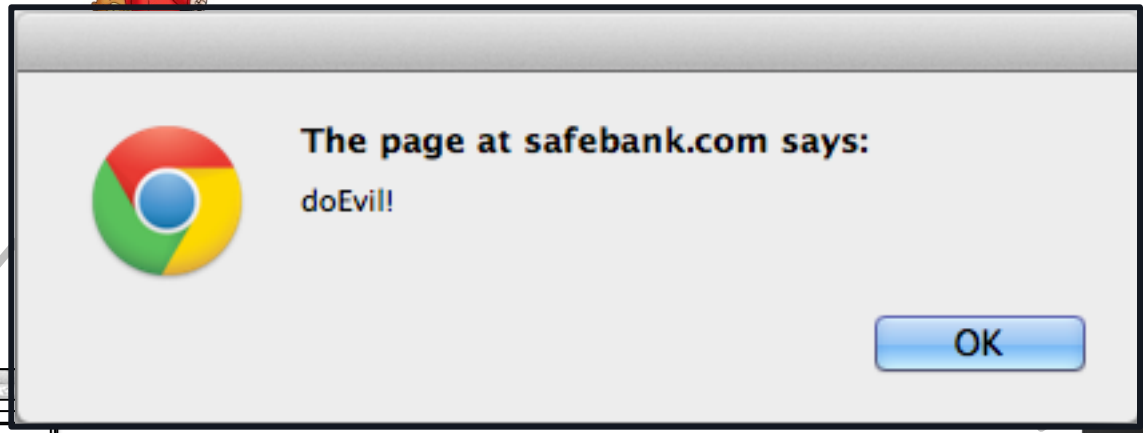
3. Server uses the params in a safe fashion, or ignores the malicious param



4. Safe HTML

Vulnerable Server

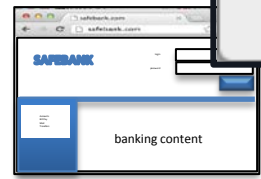
# DOM-Based XSS



uses the params in a  
on, or ignores the  
cious param



User



4. Safe HTML



Vulnerable Server

5. JavaScript code **ON THE CLIENT** uses the malicious params in an unsafe manner, causing code execution

# Exploiting a DOM Based XSS

- The attack payload (the URI) is still sent to the server, where it might be logged.
- In some web applications, the URI fragment is used to pass arguments
  - E.g., Gmail, Twitter, Facebook,
- Consider a more Web 2.0 version of the previous example:  
`http://example.net/welcome.php#name=Joe`
  - The browser doesn't send the fragment "#name=Joe" to the server as part of the HTTP Request
  - The same attack still exists

# Three Types of XSS

- Type 2: Persistent or Stored
  - The attack vector is stored at the server
- Type 1: Reflected
  - The attack value is 'reflected' back by the server
- Type 0: DOM Based
  - The vulnerability is in the client side code